Toward a Formal Theory of Objects

Scott Burleigh
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, mail stop 301-270
Pasadena, CA 91109
burleigh@puente.jpl.nasa.gov

Abstract

A formal statement of object model concepts is developed from simple mathematical structures. A "type" is defined as an ordered pair comprising type name and type declaration; a type declaration includes specifications of interface, design, heredity, and behavior. An "object system" is then defined as a deterministic finite automaton (DFA) specified in terms of a given "viable type" (viable types being a defined subset of types). Definitions of "object" and "class" are then developed, and such other topics as containment, message passing, inheritance (both static and dynamic), polymorphism, and persistence are discussed in this context.

Index terms: classes, finite automata, inheritance, objects, polymorphism, theory, types

Introduction

The success of object-oriented software development will be measured, in the long run, by the degree to which it reduces the net cost of developing and sustaining operational software. Those cost savings will vary with the readiness and succcss with which developers adopt the technology. This suggests that the concepts that the technology rests on ought to be easily understandable by as large and general a developer population as possible. If we want the concepts to bc easily understandable, then we need to state them clearly in familiar terms. The challenge is less to our inventiveness than to our communication skills,

Although there is now widespread and growing industry acceptance of software development on the object model, there is as yet no unambiguous and universally accepted manifesto of just what that model is [1], Thoughtful efforts to define object software concepts abound (examples include [2] and [3] among many), but most are to some degree handicapped by the imprecision and ambiguity of the English language. A more rigorous treatment is found in [4], but even these formulations stop short of a comprehensive mathematical statement of the object model.

In this paper we attempt such a statement, constructed from the simplest elements of computation theory, Our ultimate aim is the assembly of a mathematically sound theoretical framework within which the key concepts of object-oriented analysis, design, and programming can be articulated. This framework could provide a basis for examining and evaluating object software development practices, languages, and methodologies, and perhaps for devising new ones.

Very briefly, we suggest that an object system (or “object-oriented program”, or “object-oriented application”) is simply a deterministic finite automaton (DFA) that is specified in somewhat more detail than classical DFAs and is at the same time subject to certain constraints, as discussed later.

Definitions of object model concepts should generally satisfy two criteria: clarity and familiarity. The formalism of the statement offered here is intended to assure its clarity, or at least minimize its ambiguity, while the statement’s specific elements (some of which may seem arbitrary) are designed to yield as few surprises as possible in the resulting model. We are in fact striving for familiarity on several levels:

The theory, based as it is on fundamental mathematical structures, will be accessible to the undergraduate computer science student.

The formal representations of object model concepts are generally (though not always) consistent with the prevailing implementations of object technology.

Moreover, wc intend those representations to be as consistent as possible with analogous "real world" concepts that are identified by the same words.

This third point ought to be amplified somewhat. Most software developers know little about computer science. (Accountants who build spreadsheets are, after all, software developers.) But all software developers inhabit a common "real world" in which the concepts whose names have been borrowed for the object model -- class, object, message -- already have familiar meanings. Any program can be regarded as a model of some aspect of that real world, Object model concepts should enable us to discuss a program in terms that are closely analogous to those that characterize the real world system it models. If we want those concepts to be easily and accurately grasped, then we must discipline ourselves to use familiar words in familiar ways (in preference, where conflicts arise, to the ways in which they may be used in object technology as currently implemented). We believe it is a waste of cognitive leverage to stray any further from common sense than absolutely necessary.

Foundations

A DFA is fully specified by an ordered quintuple $(K, \Sigma, \delta, s, F)$ where K is a finite set of states; Σ is an alphabet; $s \in K$ is the initial state; $F \subseteq K$ is the set of final states; and δ is a function from $K \times \Sigma$ to K [5]. We extend the canonical notion of DFA in two fundamental ways:

1. Rather than simply name the states in K, we describe them exhaustively, enumerating the "elements" of each state. While it remains true that a DFA merely negotiates transitions from one state in K to another in response to input symbols, each state in K

has semantic significance that extends far beyond its name.

2. Rather than look to the final state of the DFA to learn whether or not a string of input is a member of a regular language, we set F = K to ensure that all strings of input arc accepted and instead suppose that the state transitions themselves can do useful things for us. For example, an element of a DFA's state might control the opcrat ion of a physical device (much as a hypothetical physical "tape drive" controls the delivery of input symbols to the DFA).

For convenience, we define a new binary operation on sets whose members are all sets. The *Cartesian union* of A and B where both A and B are sets of sets, denoted by A # B, is the set of all sets formed by the union of one member of A with one member of B. That is, A # B = { a *u* b : a ∈ A, b ∈ B}. If S is any set of N sets of sets, we can write #S for the set of all possible sets formed by the union of N sets Pi such that each Pi is a member of some member of S, yet no two P_is are members of the same member of S. (An invest igat ion of # operations on sets whose members are not all sets is beyond the scope of this paper.)

Let Σ be an alphabet. This alphabet wil be used in denoting the names and values of the elements of a DFA's state. Let language A be a, inite set of strings over Σ, including the empty string e and the strings *d*, which we read as "datum"; m, which we read as "method"; *a*, read as "argument"; *r*, read as "result"; ands, read as "method stack". Lct A' ⊆ A bc a sct of strings, $d \in A'$.

Wc define τ as the ordered pair $(l_\tau, (E_\tau, I_\tau, N_\tau, D_\tau, L_\tau, A_\tau, Z_\tau))$ where $l_\tau = d$, $E_\tau = \phi$ (the empty set), $I_\tau = \phi$, $N_\tau = \phi$, $D_\tau = \phi$, $L_\tau = \phi$, $A_\tau = \phi$, and $Z_\tau = \phi$. Then we define X, the set of all *types*, as a function X:A' → R where *R* is the set of all *type declarations.* Each member of X is an ordered pair $(l_X, (E_X, I_X, N_X, D_X, L_X, A_X, Z_X))$ where l_X, called the *label* of the type, is a member of A' and $(E_X$, lx, N_x, D_X, LX, Ax, $Z_X)$ is a type declaration defined as follows:

- E_x, called the *characteristic method list* of the type, is a subset of A such that e ∉ N_x.

- lx, the *characteristic infer-ice* of X, is a function from 13_x to *T* x *T* where *T is* the set of all *viable types*. (We define "viable type" later; for now, we note only that *T* ⊆ *X.) lx* is thus a set of ordered pairs (m, (T_a, T_r)) such that m ∈ E_x is the name of a method, T_a is the type of the method's argument, and T_r is the type of the method's result. lx in effect characterizes all the methods X is capable of executing.

The definition of Ix enables us to define two other terms that will bc useful later:

The *argument parameter type set* of X, denoted by AP_x, is defined by AP_x={T : (m, (T, T,))= lx}.

The *result parameter type set* of X, denoted by RP_x, is defined by RPx = {T : (m, (T_a, T)) ● IX}.

- N_x, called the *characteristic name space* of X, is a subset of A constrained as follows:

  a. *e* ∉ N_X

  b. No member of N_x may ever have m, a, *r,* ors as a prefix.

- Dx, called the *characteristic design* of the type, is a function D_x: $N_x \rightarrow$ *T*. Each member of function D_x, called an *attribute, is* an ordered pair (n, T) where T is a viable type and n, the *name* of the attribute, is a string n ● N_x.

The definition of D_x gives us a basis for defining some additional terms:

The *implicit design* of X, denoted by Ax, is defined as follows:

If X = τ, then Ax = φ.

Otherwise, Ax= D_x *u*{ (*m*, τ), (*s*, τ)} *u* { (*a*l_T, T) : T ∈ AP_x}∪ {(*r*l_T, T) : T ∈ **RPx }.** [*a*l_T is the string formed by concatenating the string a ("argument") with the label of class T, which is also a string. rl_T is formed similarly.]

The implicit design of a type includes all elements of the type's characteristic design together with those attributes that are implicitly involved in the behavior specified by the type: method, method stack, and all "argument" and "result" attributes alluded to in the type's interface.

The *characteristic dominion* of X, denoted by G_x, is defined as follows:

If $X = \tau$, then $G_x = \phi$.

Otherwise, $G_x = \{n : (n, \tau) \in Ax\}$ -- that is, the names of all members of X's implicit design that are of type τ.

The *characteristic state space* of X, denoted by K_x, is defined as follows:

If $X = \tau$, then $K_x = \phi$.

Otherwise, $K_X = (\#\{B: B = n \times \Lambda, n \in G_X - \{m\}\}) \# (m \times E_X\})$ -- that is, the Cartesian **union** of all Cartesian **products** obtained by crossing the names in the characteristic method list of X with the name m, and the language A with all other members of X's characteristic dominion.

The *qualified name space* of an attribute (n, T) in a characteristic design D_X, denoted by $R_{(n, T)}$, is defined as follows:

If $T = \tau$, then $R_{(n, T)} = \phi$.

Otherwise, $R_{(n, T)} = \{ nw : w \in \mathbf{N}_T)$ where NT is the *aggregate* *name space* of T, which we will define shortly.

In other words, wc get the qualified name space of an attribute by prepending the attribute's name to all the names in the aggregate name space of the attribute's type.

The *qualified design* of an attribute (n, T) in a characteristic design D_X, denoted by $Q_{(n, 1)}$, is defined as follows:

If $T = \tau$, then $Q_{(n, T)} = \phi$.

Otherwise, $Q_{(n, T)} = \{ (nw, U) : (w, U) \in \mathbf{D}_T \}$ where $\mathbf{D}_T$ is the *aggregate design* of T, which, again, we will define shortly.

That is, we get the qualified design of an attribute by prepending the attribute's name to the names of all attributes in the aggregate design of that attribute's type.

- Lx, called the *linkage* of the type, is defined by $L_X \subseteq \{(p, q) : p \in (\{n : (n, T) \in D_X\} \cup \{n_1 n_2 : (n_1, U) \in D_X, (n_2, T) \in D_U,\}), q \in \{n_1 n_2 : (n_1, V) \in D_X, (n_2, T) \in \Delta V, n_2 \in \{m, a l_T, r l_T\}\}$ subject to the following constraints:

  a. If $(pi, q_i) \in$ Lx then $(p_j, q_i) \notin L_X$ for all $p_j \neq p_i$ and $(p_i, q_j) \notin L_X$ for all $q_j \neq q_i$. That is, no two p's maybe linked to the same q, and no p may be linked to more than one q.

  b. If $(n_2, n_1 a l_T) \in$ Lx where $(n_2, T) \in D_x$, then $(n_3, n_1 m) \in I._x$ for some $(n_3, \tau) \in D_x$. Likewise, if $(n_4 n_2, n_1 a l_T) \in$ Lx where $(n_4, U) \in D_x$ and $(n_2, T) \in D_U$, then $(n_4 n_3, n_1 m) \in$ Lx for some $(n_3, \tau) \in D_U$. In short, whenever any attribute in a given design is linked to any "argument" of some other element of that design, a third attribute (of type τ) in the same design must be linked to the "method" of that element.

In other words, any attribute in the characteristic (not implicit) design of type X maybe linked to the method or any argument or result attribute <u>of the same type</u> in the implicit design Of any non-τ attribute of X, as may any attribute in the characteristic design of any non-τ attribute of X, As we explain later, linkage provides the mechanism for communication between pairs of objects in an object system.

The definition of $l._X$ gives us a basis for an additional definition:

> The *qualified linkage* of an attribute (n, T) in a characteristic design D_X, denoted by $J_{(n, T)}$, is defined as follows:
>
> > If $T = \tau$, then $J_{(n, T)} = \phi$.
> >
> > Otherwise, $J_{(n, T)} = \{ (\text{rip}, \text{nq}) : (p, q) \in L_T \}$ where L_T is the *aggregate linkage* of T, which we wil 1 define soon.
> >
> > In a by-now familiar fashion, we get the qualified linkage of an attribute by prepending the attribute’s name to all the names in the aggregate linkage of the attribute’s type.

- Ax, called the *characteristic ancestry* of the type, is a set of zero or more sets called *ancestors;* each ancestor is a set of one or more types, referred to as *personae.* The ancestry of X enumerates those types of which X can be considered a refinement or qualification, and from which X can be said to “inherit” capability and behavior.

Ax is constrained in several ways, one of which must be stated before we go on: al of the personae of any ancestor must have identical characteristic method list, interface, and ancestry. The characteristic method list common to all members of an ancestor H (the *common method list* of H) is denoted by CE_{11}. The characteristic interface common to all members of an ancestor H (the *common interface* of H) is denoted by CI_H. The characteristic ancestry common to all members of an ancestor H (the *common ancestry* of H) is denoted by CA11.

The definition of Ax gives us a basis for further definitions:

The *aggregate common ancestry* of an ancestor H, denoted by **CA** $_{H}$, is defined by *(u {CAY:* Y ∈ CA_H }) u { CA_H }. That is, the aggregate common ancestry of an ancestor H (which is a set oft ypcs) contains H's own common ancestry and the aggregate common ancestries of all members of that common ancestry. Note that |$\mathbf{CA}_H$| might not be equal to |CA_H| + the sum of | CA_Y | over all Y ∈ CA_H because two or more members of CA_H might have a common ancestor. By taking the union of aggregate ancestries we eliminate redundant ancestors.

The *aggregate ancestry* of X, denoted by Ax, is defined as Ax = $CA_{\{X\}}$. We get the aggregate ancestry of type X, in other words, by considering {X} as an ancestor and computing that set's aggregate common ancestry. Like Ax, Ax is a set of zero or more different sets of types, each characterized by an interface.

In similar fashion, the *aggregate common method list* of an ancestor H, denoted by $\mathbf{CE}_H$, is defined by *(u {* $\mathbf{CE}_Y$: Y ● $CA_{}$ }) *u {* CE_H} and the *aggregate method list* of X, denoted by $\mathbf{E}_X$, **is** defined as $\mathbf{E}_X = \mathbf{CE}_{\{X\}}$; the *aggregate common interface* of an ancestor H, denoted by CI_H, is defined by *(u {* CI_Y: Y ● CA_H }) *u {* CI_H) and the *aggregate interface* of X, denoted by **lx, is** defined as $\mathbf{I}_X = \mathbf{CI}_{\{X\}}$.

The *characteristic personality* of X -- the set of all types cited in the characteristic ancestry of X, denoted by P_x -- is defined as P_x = UA_X.

The *aggregate personality* of X, denoted by P_x, is defined as $\mathbf{P}_X = (\cup \{P_y : Y \in P_x\}) \cup \{X\}$. That is, the aggregate personality of X contains X itself and the aggregate personality ies of al 1 members of $\mathbf{P}_X$. Note that $|P_x|$ might not be equal to 1 + the sum of $|P_y|$ over all $Y \in P_x$ because two or more members of P_x might have the same ancestor. By taking the union of aggregate personalities wc eliminate redundant types in the aggregate personality of X,

The *aggregate name space* of X, denoted by N_x, is defined by $N_x = \cup \{s : s = (\cup\{R_{(n, T)} : (n, T) \in D_Y\} \cup N_y), Y \in P_x)$. That is, the aggregate name space of a type X is the union, over all members Y of P_x, of the characteristic name space of Y together with the qualified name spaces of all attributes in the characteristic design of Y.

The *aggregate design* of X, denoted by $\mathbf{D}_X$, is defined by $\mathbf{D}_X = \cup\{s : s = (\cup\{Q_{(n, T)} : (n, T) \in \Delta_Y\} \cup A_y), Y \bullet \mathbf{P}_X\}$. That is, the aggregate design of a type X is the union, over all members Y of P_x, of the implicit design of Y together with the qualified designs of all attributes in Y's implicit design.

The *aggregate state space* of X, denoted by K_x, is defined as:

$$\mathbf{K}_X = \#\{B : B = n \times \Lambda, (n, \tau) \in \mathbf{D}_X\}$$

-- that is, the Cartesian **union** of the Cartesian **products** obtained by crossing the language A with the fully qualified names of all attributes of type τ in the aggregate design of X. Each member of the aggregate state space of type X -- each *state* of X -- is a set of ordered pairs (n, v) where n is a fully qualified name and v is a "value" -- a string -- selected from A. Each such ordered pair in any such state is called an *element* of that state.

The *aggregate linkage* of X, denoted by l_X, is defined by $I_x = \cup\{s : s = (\cup\{J_{(n,T)} : (n, T) \in \Delta_Y\} \cup L_Y), Y \in P_X\}$. That is, the aggregate linkage of a type X is the union, over all members Y of P_x, of the characteristic linkage of Y together with the qualified linkages of all attributes in the implicit design of Y.

The *contradiction set* of a state $k \in K_x$, denoted by C_k, is defined as:

$$C_k = \{(n_i, v_i) : (n_i, v_i) \bullet k, \ (n_i, n_j) \in L_X, (n_j, v_j) \in k, v_i \neq v_j\} \ .$$

(Non-null contradiction sets occur when conflicting values for two equivalent [linked] attributes result from the Cartesian union of the Cartesian products of the two attributes.)

The *resolved state space* of X, denoted by p_x, is defined as:

$$\rho_X = \{k : k \in K_X, \ C_k = \phi\} \quad .$$

The resolved state space of X is thus the set of all possible states of X that contain no contradictions resulting from linkage.

The *current method* of a given state $k \in p_x$, denoted by c_k, is the value of that member of k whose fully qualified name is m.

For now, in the interest of brevity and clarity, wc constrain ancestors as follows: no ancestor may have more than one persona. (Multiplicity of personae will eventually enable us to develop a theoretical framework for such concepts as fuzzy and dynamic inheritance, but it adds considerable complexity to the model. We therefore defer discussion of this topic until a later paper, though we do hint at it later in this one.) Ax is further constrained as follows:

a. For all $Y \in P_x$: $X \notin P_y$ and for all $(n, T) \in D_y$, $\mathbf{T \# X}$. That is, X must not be a member of the aggregate personality of any member of its own characteristic personality, and no attribute of type X maybe a member of the aggregate design of any member of X‘s characteristic personality y.

b. For all (n, T) ● D_X: T # X and for all Y ∈ P_T, X ∉ P_y. No attribute of type X maybe a member of X‘s aggregate design, and X must not be a member of the aggregate personality of any member of X’s aggregate design.

c. $N_x \subseteq$ A (and therefore N_x must be finite).

d. |Nx| must be equal to the sum of ($|N_y|$ plus the sum of $|R_{(n, T)}|$ over all (n, T) ● D_y), over all Y ∈ P_x. That is, the sets of qualified attribute names contributed by all members of the aggregate personality of X must be disjoint.

e. $|E_X|$ must be equal to $|13_x|$ + the sum of $|CE_H|$ over all H ● Ax. I.e., the method lists of all of X’s ancestors and of X itself must be disjoint.

- Z_X, the *characteristic transition function* of the type X, is a function Z_X: K_x x Σ → #{B: B = n x A, n ∈ G_x}. Every member t ∈ Z_X, called a *transition* in Z_X, is an ordered pair ((j, b), k) where j ● K_x -- i.e., is a state of X [a set of (name, string) ordered pairs] in which m ∈ E_x -- and b ● Σ, and k is a member of a state space that is identical to K_x except that in its members *m* can have any value whatsoever (selected from A). Note that this means that a transition in Z_X can result in a state that is outside the characteristic state space of X, i.e., one to which X’s characteristic transition function cannot be applied.

The members ((j, b), k) of Z_X are constrained as follows:

a. Given (n], T) ● D_X, T ≠ τ, and (n_2, τ) ∈ D_X and (n_2, n_1*ad*) ● Lx [implying, by the constraints on Lx, that (n_3, n_1*m*) ∈ Lx] and (n_2, w]) ∈ j and (n_3, W_2) ● j: if w_2 ≠ e then (n_2, WI) ∈ k and (n_3, W_2) ● k. That is, the values of attributes linked either to the method or to any argument of type τ of some other attribute cannot change unless that attribute's method is null,

b. If (n], T) $\in D_X$, $T \neq \tau$, and $(n_2, \tau) \in D_X$ and $(n_2, n_1 rd) \in Lx$ and $(n_2, w_1) \in j$ then $(n_2, w_1) \bullet k$. (No transition may specify a change in the value of an attribute linked to any result of type τ of another attribute,)

From this definition of Z_X we can develop three additional concepts:

The *qualified transition function* of an attribute (n, T) in a design D_x, denoted by $M_{(n, T)}$, is defined as follows:

If $T = \tau$, then $M_{(n, T)} = \phi$.

Otherwise, $M_{(n, T)} = \{((j, b), k) : j = \{(nn_1, w_1) : (n_1, w_1) \in j_\mu\}, k = \{(nn_2, w_2) : (n_2, w_2) \in k_\mu\}, ((j_\mu, b), k_\mu) \bullet \mu_T\}$ where μ_T is the *resolved transition function* of T, defined below.

informally, wc get the qualified transition function of an attribute by prepending the attribute's name to the attribute names of the elements of the "before" and "after" states of each transition in the resolved transition function of the attribute's type.

The *aggregate transition function* of type X, denoted by Z_X, *is* defined by:

$Z_X = \{((j, b), k) : j \in \rho_X, b \in \Sigma, k =$

u $\{M_{(g, T)}(\{(n, v) : (n, v) \in j, (n, \tau) \in Q_{(g, T)}\}, b) : (g, T) \in \cup\{D_V : V \in P_X\}, T \neq \tau\}$

u $Z_X(\{(n, v) : (n, v) \in j, n \in G_y$ where Y is the member of P_x such that $c_j \in E_Y\}, b)$

u $\{(n, v) : (n, v) \bullet j, n \in G_w$ where W is <u>not</u> the member of P_x

such that $c_j \in E_Y$, and $n \notin G_Y$ where Y is the member of P_x such that $c_j \bullet E_Y$ }.

Put another way, each transition in function Z_X is from a state j to a state k which is identical to j except that:

a. If the current method of j is a member of the characteristic method list of any type Y in X's aggregate personality, then the values of all members of the characteristic dominion of Y are modified according to Y's characteristic transition function.

b. The values of all state elements that are not in the aggregate dominion of X (that is, whose names are the names of attributes of type other than τ in the characteristic designs of members of X's aggregate personality) are modified according to the qualified transition functions of the associated types. For each such design member, the qualified transition function is exercised on the set of all state elements in j whose names are those of attributes of type τ in the qualified design corresponding to that function.

The *resolved transition function* of a type X, denoted by μ_X, is a transformat ion of X's aggregate transition function that resolves possible conflicts arising from the aggregate linkage of X:

For each ((j, b), k) in Z_X:

For each $(n_2, n_1 m) \in I._x$:

Clearly both $(n_2, W_2) \in k$ and $(n_1 m, w_1) \in k$.

Since n_2 and $n_1 m$ are equivalent, (n_2, w_2) and $(n_1 m, w_1)$ must agrec.

(Moreover, if $(n_3, n_1 ad) \in Lx$ then $(n_3, w_3) \in k$ and $(n_1 ad, w_4) \in k$, and (n_3, w_3) and $(n_1 ad, w_4)$ must agree,)

If $w_1 = e$, then wc replace $(n_1 m, w_1)$ ink with $(n_1 m, W_2)$; also, if $(n_3, n_1 ad) \in I._x$, we replace $(n_1 ad, W_4)$ with $(n_1 ad, w_3)$ in k.

Otherwise, we replace (n_2, W_2) in k with (n_2, WI); if $(n_3, n_1 ad) \in l._x$, we replace (n_3, w_3) with (n_3, W_4) in k.

For each $(n_2, n_1 rd) \in J,_x$:

Clearly both $(n_2, W_2) \in k$ and $(n_1 rd, w_1) \in k$.

Since n_2 and $n_1 rd$ arc equivalent, (n_2, W_2) and $(n_1 rd, w_1)$ must agree.

Therefore we replace (n_2, W_2) with (n_2, w_1) in k.

In essence, whenever the “target” attribute is active -- has non-null method -- its type’s transition function prevails over X’s transition function (which, as it is constrained, couldn’t effect any change in the method or argument or result of the attribute anyway), At other times (when, again, the constraints on characteristic transition functions preclude any change in any data of thc target attribute), X’s transition function prevails .

With the set of all types defined, we can define *T*, the set of all *viable types,* as *T= {X : X* $\in$ *X*,

$\{k : ((j, b), k) \in \mu_X, c_k \notin (E_X \cup \{e\})\} = \phi\}$. That is, a viable type is one whose resolved transition function never puts it into a state in which its current method is neither e nor one of the members of its aggregate method list. Because $\mu_\tau = \phi$, τ is trivially a viable type.

We can now define an *object* system of type T, where $T \in 1$, as any DFA $(\rho_T, \Sigma, \mu_T, s, \rho_T)$ such that $s \in \rho_T$.

Given an object system of type T and a name n such that $(n, X) \in D_T$, we define the *object named by n*, denoted by g_n, as $\{ nw : w \in A, nw \in NT\}$. We say that the name of g_n is n and that the type of g_n is X (i.e., that g_n is an object of type X or, informally, that n is an object of type X).

Given any state $k \in \rho_T$ of this object system, the state of object g_n when the system is in that state is defined as $\{(n_i, vi) : (n_i, vi) \in k, n_i \in g_n\}$. The behavior of object g_n is $\{((j, b), k) : ((j, b), k) \in \mu_T, (rim, w) \in j, w \neq e \}$, the set of all state transitions in the resolved transition function of T such that the value of the state element corresponding to the method of object g_n in the initial state is not the null string.

If the names of objects g_A and g_B are the names of two members of the characteristic (not aggregate) design of T, then g_A and g_B are each other's peers; if g_A is an object of type U and objects g_D and g_E correspond to members D and E of the characteristic design of U, then g_D and g_E are *immediately contained* in g_A (g_A is their *immediate container)* and they are each other's peers, but neither is a peer of either g_A or g_B.

Finally, given an object system of type T and a type X, we define the *class* of X (C_x, or "the class X") as the set of all objects g_n of any type U such that $X \in P_u$. The behavior of each such object will include the behavior of objects of type X, since the aggregate method list of any U is a superset of the characteristic method list of X, and therefore the aggregate transition function of U will behave according to X's characteristic transition function whenever the current method is one defined in X's characteristic method list. A class C_w is a subclass of class C_X if and on] y if every member of C_W is also a member of C_X -- i.e., $C_w \subseteq C_x$ -- or, equivalently, $X \in I'_w$.

Examples

Suppose we stipulate that the symbol O (where $O \in \Sigma$) signifies "no meaningful input" or "end of input string". We could define a type "register" by **(register, (E,** 1, N, D, **L, A,** Z)) where:

E = {log, dump}

I = (log, (τ, τ)), (dump, (τ, τ))}

N = {value)

D = { (value, τ) }

L = ϕ

A = ϕ

Z = {(({(value, w), (*m*, log), *(ad, x)*, (*rd*, *y), (s, z))*, a),

{(value, wa), (m, log), *(ad, x)*, (*rd*, *y), (s, z)}) :* $a \in \Sigma$, $a \neq \Diamond$, $w \in A$, x ● A, $y \in A$, $z \in$ A)

$\cup$ {((((value, w), (*m*, log), (*ad*, x), (*rd*, y), (*s*, z)}, a),

{(value, w), (*m*, *e*), (*ad*, x), (*rd*, y), (s, z)}) : a =0, $w \in L$, x ● L, y ● L, $z \in L$}

u {(({(value, w), (*m*, dump), (*ad*, x), (*rd*, y), (*s*, z)}, a),

{(value, e), (*m*, e), *(ad, x)*, (*rd*, *w), (s, z)})*: $a \in \Sigma$, $w \in$ **L, x** $\in$ **L, y** $\in$ **L,** $z \in$ **L}**

A register's "log" method merely acquires and stores all input symbols until the O symbol, appending them to its value, then stops. Its "dump" method copies its value to its result attribute and crases the value.

For the next definition we rely on the natural isomorphism between the set of integers and the set I of all symbolic representations of integers in a given alphabet (Σ in this case), $I \subseteq A$. Given this, wc can define the type “integer” by **(integer, (E, J, N,** D, **L, A,** Z)) where:

$E = \{\text{load}, ?\}$

$1 = \{(\text{load}, (\tau, \tau)), (?, (\tau, \tau))\}$

$N = \{ \text{value} \}$

$\mathbf{D = \{}\ (\text{value}, \tau)\ \}$

$L = \phi$

$A = \phi$

$Z = \{\ \{((\{(\text{value}, w), (m, \text{load}), (ad, x), (rd, y), (s, z)\}, a),$

$\{(\text{value}, w), (m, e), (ad, e), (rd, e), (s, z))) : a \in \Sigma, w \bullet A, x \in A, x \notin I, y \in A, z \in A]$

$u\ \ \{((\{(\text{value}, w), (m, \text{load}), (ad, x), (rd, y), (s, z)\}, a),$

$\{(\text{value}, w), (m, e), (ad, e), (rd, x), (s, z)\}): a \in \Sigma, w \bullet A, x \in I, y \in A, z \in A\}$

$\cup\ \ \{((\{(\text{value}, w), (m, ?), (ad, x), (rd, y), (s, z)\}, a),$

$\{(\text{value}, w), (m, e), (ad, e), (rd, w), (s, z)\}) : a \in \Sigma, w \in A, x \in A, y \in A, z \in A\}$

An integer can receive a ncw value via its load method, but rejects all new values that are not symbolic representations of integers; therefore its value is guaranteed at all times to be either the nul 1 string or the symbolic representation of an integer. It reports its current value when its ? method is triggered.

Now we could define an additional type “number” that can hold integer values and add integer values to them. For this definition wc assume that the strings +, 1, 2, and 3 are members of Λ. The definition might look like **(number, (E, I,** N, D, L, A, Z)) where:

E= {+}

I = {(+, (integer, integer))}

N = { phase, integerVal)

D = { (phase, τ), ((integerVal, integer), (vmethod, τ), (vresult, τ), (amethod, τ), (aresult, τ), (rmethod, τ), (rarg, τ) }

L = { (vmethod, integerVal*m*), (vresult, integerVal*rd*), (amethod, ***a*integer*m***), (aresult, ***a*integer*rd***), (rmethod, ***r*integer*m***), (rarg, ***r*integer*ad***) **}**

A = ϕ

Z = { {(({(w, +), (vmethod, p), (vresult, q), (amethod, r), (aresult, s), (rmethod, t), (rarg, u), (s, z), (phase, v)], a),

{(m, +), (vmethod, p), (vresult, q), (amethod, r), (aresult, s), (rmethod, t), (rarg, u), (*s*, z), (phase, v)}) : p $\in \Lambda$, p $\neq$ *e*, q $\in \Lambda$, r $\in \Lambda$, s $\in \Lambda$, t $\in \Lambda$, u $\in \Lambda$, z $\in \Lambda$, v $\in \Lambda$,

v $\notin$ {1,2,3}}

-- wait until integerVal is inactive

u {{(({ (*m*, +), (vmethod, *e*), (vresult, q), (amethod, r), (aresult, s), (rmethod, t), (rarg, u), (*s*, z), (phase, v)}, a),

{(m, +), (vmethod, ?), (vresult, q), (amethod, r), (aresult, s), (rmethod, t), (rarg, u), (*s*, z), (phase,)}): q $\in \Lambda$, r $\in \Lambda$, s $\in \Lambda$, t $\in \Lambda$, u $\in \Lambda$, z $\in \Lambda$, v $\notin$ {1,2,3)}

-- then ask for the value of integerVal

u {{(({ (*m*, +), (vmethod, ?), (vresult, q), (amethod, r), (aresult, s), (rmethod, t), (rarg, u), (*s*, z), (phase, 1)}, a),

{(m, +), (vmethod, ?), (vresult, q), (amethod, r), (aresult, s), (rmethod, t), (rarg, u),

(s, z), (phase, l)}): $q \in \Lambda, r \in \Lambda, s \in \Lambda, t \in \Lambda, u \in \Lambda, z \in \Lambda\}$

-- wait until it's delivered

u {{(({(m, +), (vmethod, p), (vresult, q), (amethod, r), (aresult, s), (rmethod, t), (rarg, u),

(s, 7.), (phase, 1)}, a),

{(m, +), (vmethod, p), (vresult, q), (amethod, r), (aresult, s), (rmethod, t), (rarg, u),

(s, z), (phase, l)}): $p \in \Lambda, p \neq ?, q = e, r \in \Lambda, s \in \Lambda, t \in \Lambda, t \neq e, u \in \Lambda, z \in \Lambda\}$

-- if it's null, wait until ***r*integer** (the return value for the method) is inactive

u {{(({(WI>+), (vmethod, p), (vresult, q), (amethod, r), (aresult, s), (rmethod, *e*), (rarg, u),

(s, z), (phase, l)), a),

{ (*m*, +), (vmethod, p), (vresult, q), (amethod, r), (aresult, s), (rmethod, load),

(rarg, e), (s, z), (phase, 3)}) : $p \in \Lambda, p \neq ?, q = e, r \in \Lambda, s \in \Lambda, u \in \Lambda, z \in \Lambda\}$

-- ...and then put the null string into it...

u {{(({ (*m*, +), (vmethod, p), (vresult, q), (amethod, r), (aresult, s), (rmethod, load), (rarg, u),

(s, z), (phase, 3)), a),

{(m, +), (vmethod, p), (vresult, q), (amethod, r), (aresult, s), (rmethod, load), (rarg, u),

(s, z), (phase, 3)}) : $p \in \Lambda, q \in \Lambda, r \in \Lambda, s \in \Lambda, u \in \Lambda, z \in \Lambda\}$

--...wait until it's accepted...

u {{(({(711, +), (vmethod, p), (vresult, q), (amethod, r), (aresult, s), (rmethod, t), (rarg, u),

(s, z), (phase, 3)}, a),

{(m, e), (vmethod, p), (vresult, q), (amethod, r), (aresult, s), (rmethod, t), (rarg, u), (s, z), (phase, *e*)}) : p ∈ Λ, q ∈ Λ, r ∈ Λ, s ∈ Λ, t ∈ Λ, t ≠ load, u ∈ Λ, z ∈ Λ}

--... and de-activate the number.

u { *{(({(m, +),* (vmethod, p), (vresult, q), (amethod, r), (aresult, s), (rmethod, t), (rarg, u), (s, z), (phase, 1)}, a),

{(m, +), (vmethod, p), (vresult, q), (amethod, r), (aresult, s), (rmethod, t), (rarg, u), (s, z), (phase, 1)}) : p ∈ Λ, p ≠ ?, q ∈ Λ, q ≠ *e*, r ∈ Λ, r ≠ *e*, s ∈ Λ, t ∈ Λ, u ∈ Λ, z ∈ Λ}

--otherwise, wait until *a***integer** (the **argument** passed to the method) is inactive

u { *{(({(m,* +), (vmethod, p), (vresult, q), (amethod, *e*), (aresult, s), (rmethod, t), (rarg, u), (s, z), (phase, 1)}, a),

{(m, +), (vmethod, p), (vresult, q), (amethod, ?), (aresult, s), (rmethod, t), (rarg, u), (s, z), (phase, 2)}) : p ∈ Λ, p ≠ ?, q ∈ Λ, q ≠ *e*, s ∈ Λ, t ∈ Λ, u ∈ Λ, z ∈ Λ}

--then ask for its value

u { {(({(m, +), (vmethod, p), (vresult, q), (amethod, ?), (aresult, s), (rmethod, t), (rarg, u), (s, z), (phase, 2)}, a),

{(m, +), (vmethod, p), (vresult, q), (amethod, ?), (aresult, s), (rmethod, t), (rarg, u), (s, z), (phase, 2)}) : p ∈ Λ, q ∈ Λ, s ∈ Λ, t ∈ Λ, u ∈ Λ, z ∈ Λ}

--wait until it’s delivered

u ({(({(m, +), (vmethod, p), (vresult, q), (amethod, r), (aresult, s), (rmethod, t), (rarg, u), (s, z), (phase, 2)}, a), {(m, +), (vmethod, p), (vresult, q), (amethod, r), (aresult, s), (rmethod, t), (rarg, u),

(s, z), (phase, 2)}) : $p \in \Lambda$, $q \in \Lambda$, $r \in \Lambda$, $r \neq ?$, $s \in \Lambda$, $t \in \Lambda$, $t \neq e$, $u \in \Lambda$, $z \in \Lambda$}

--then wai until ***r*integer** (the return value for the method) is inactive

u { {(({(m, +), (vmethod, p), (vresult, q), (amethod, r), (aresult, e), (rmethod, e), (rarg, u),

(s, z), (phase, 2)}, a),

{(m, +), (vmethod, p), (vresult, q), (amethod, r), (aresult, e), (rmethod, load),

(rarg, e), (s, z), (phase, 3)}) : $p \in \Lambda$, $q \in \Lambda$, $r \in \Lambda$, $r \neq ?$, $u \in \Lambda$, $z \in \Lambda$}

-- if argument value is null, put the null string into it

--(thus entering same state as when integerVal was the null string)

u { {(({(m, +), (vmethod, p), (vresult, q), (amethod, r), (aresult, s), (rmethod, e), (rarg, u),

(s, z), (phase, 2)}, a),

{ (m, +), (vmethod, p), (vresult, q), (amethod, r), (aresult, s), (rmethod, load),

(rarg, q + s), (s, z), (phase, 3)}) : $p \in \Lambda$, $q \in \Lambda$, $r \in \Lambda$, $r \neq ?$, $s \in \Lambda$, $s \neq e$,

$u \in \Lambda$, $z \in \Lambda$}

--otherwise, put the sum of the value of integerVal and the value of ***a*integer** into it

--(again, entering same state as when integerVal was the null string)

--("q + s" denotes the symbolic representation of the sum of the two integers whose symbolic

representations are q and s)

In greatly compressed form:

```
If m = "+":
    If phase= 1:
        If vmethod = ?
        No change in state.
    Else
        If vresult = e
            If rmethod # e
                No change in state.
            Else
                Change rmethod to load, rresult to e, phase to 3.
        Else
            If amethod # e
                No change in state.
            Else
                Change amethod to ?, phase to 2.
Else if phase= 2:
    If amethod = ?
        No change in state.
    Else
        If rmethod # e
            No change in state.
        Else
            If aresult = e
                Change rmethod to load, rresult to e, phase to 3.
            Else
                Change rmethod to load, rresult to vresult + aresult,
                phase to 3.
Else if phase= 3:
    If rmethod = load
        No change in state.
    Else
        Change m to e, phase toe.
Else
    If vmethod # e
        No change in state.
    Else
        Change vmethod to ?, phase to 1.
```

A number is passed an integer, rather than string, argument. Addition of an integer to a number is a three-phase process, as indicated by the values that the “phase” attribute of the type can assume in performing an addition. First the number uses the ? method of its own integerVal attribute (also an integer) to obtain that value; then it uses the ? method of the argument to obtain the value of the argument; finally it uses the load method of the integer result in its interface to load the sum of these two values into that result. When either operand is the null string, the result is the null string indicating an error. No other checking for operand validity is necessary because the operands are both guaranteed to be valid integers if not null. The encapsulation of the operands is never compromised.

The phase attribute can be thought of as the name of an internal function that is called from within the execution of the method. All methods are public, but such mechanisms as phase attributes can be used to privately modularize processing that might be common to multiple methods or to multiple phases of the execution of a single method.

A number could have additional value attributes of other numeric types, such as real and imaginary, and could have additional interface elements for adding different types of numbers -- and for performing other arithmetic operations such as assignment, subtraction, etc. where the operands could be of any combination of types. The methods of number could determine which argument and which of its own values to use in a given computation by looking for the ones with non-*e* value. (E.g., non-e values for both real and imaginary attributes would signify a complex number.)

Discussion

We believe these definitions model the specification and execution of object software with quite high fidelity. Objects have identities (unique names), types, state, and behavior; classes are sets of objects that share some well-defined characteristics. Everything that can "have a value" is an object. Objects of type τ have values of an extremely simple data type, while objects of other types cent ain other, simpler objects (i .e., arc literal] y supersets of them) and can therefore be thought of as having values of arbitrarily complex abstract data types; the objects contained in a given object can be thought of as its "slots" or *instance variables.*

The arrival of input symbols models the passage of time, and the manner in which the state of an object system changes over time (as symbols arrive) is dictated by a transition function that convincingly simulates causation. Thus, through the linkage mechanism, the state of one object can be thought of as "causing" changes in the states of other objects; this in turn can be viewed as flow -- or communication -- of control and data among objects in the system. An object whose *m* instance variable has the value e in a given state can easily be considered "inactive" in that state, in the sense that the values of its other instance variables of type τ remain unchanged over time (i.e., in the course of the next state transition) unless modified by linkage from the object's immediate container or a peer. This in turn invites us to think of a change in the value of an object's m instance variable from *e* to some non-empty string as an "activation" of that object -- in other words, the initiation of the method whose name is the new value of the *m* instance variable.

It might reasonably be asked why a theory of objects should be based on deterministic finite automata rather than more powerful structures, such as Turing machines. Our answer is that DFAs arc extremely simple and easily grasped, and they seem to be sufficient. The goal of this work, after all, was to develop a formal model of object-oriented software for digital computers, which is a finite problem domain. The computations we undertake using objects might well involve values of infinite precision, such as 1/3; however, whenever we use a digital computer to perform any such computat ion wc can only use finite representations of these values, either approximations (such as .3333333333 for 1/3) or symbols (such as "1/3", π, etc.), both of which take the form of finite-length strings of symbols from some alphabet. Therefore the number of objects we use and the number of possible states each object can actually enter necessarily remain finite.

"Finite" need not, of course, mean "small". Even an object g_n of the simplest type, τ, will have as many possible states as the number of strings in A; if A is limited to "ASCII strings of fewer than 256 symbols", g_n can enter any of 2^{1792} states. The aggregate transition function for the most trivial type of which g_n is an attribute is certain to be very large, but still finite.

Parallelism and distribution

All objects other than those of type τ have m instance variables and can therefore, potentially, be concurrently active. That is, this model of object software execution is inherently parallel, with the limit on the parallelization of a given object system being one processor per non-τ object, As such, the model encourages object software technology that exploits massively parallel processing architectures. Implementation on non-parallel hardware of software built on this model would involve de-parallelizing it, e.g., in a manner analogous to multitasking on single-processor computers.

As it is inherently parallel, software built on this model is also inherently distributable. Given an object system of type T, not only could all non-τ objects in the characteristic design of **T** conceivably be instantiated on different machines (the familiar "distributed object system" approach, where "top-level" objects are typical] y implemented as operating system processes, tasks, or threads) but also the objects contained in any such object could be similarly distributed, making that object a true "distributed object".

Object interaction

The decision to support exhaustive parallelization and object distribution, together with a commitment to bulletproof encapsulation of object data, dictated the constraints imposed on linkage within a type.

Data can be inserted into or extracted from an object of t ypc other than τ (i.e., the object's state may bc altered or interrogated) only by initiating the methods defined in the method list of that object's type -- that is, by causing the object's method instance variable and, possibly, one of its argument instance variables to change value and then examining one of its result instance variables. (Strictly speaking, only a single argument per method invocation is supported. However, this argument could be of any arbitrarily complex type and therefore could be of a type that includes any number of unrelated argument objects of various types; explicit multiplicity of method arguments might well be offered by a specific language implementation of this model, but it is not an object model concept.)

This method initiation scheme relies completely on the linkage of the invoking object's state elements to those of the t arget object, The members of the characteristic design of an object g_B are invisible to the transition function of every other object in the object system, including the object g_A within which g_B is immediately contained; that is, g_B is fully encapsulated. The type specification of which g_B is a characteristic design element can, however, link other members of its own characteristic design to the method and arguments of g_A (as enumerated in g_A's aggregate interface) and can similarly link members of the characteristic design of object g_C, which is a peer of g_A. No other object g_H in the object system can have any direct access whatsoever to g_B: if g_H were a peer or container of g_A, then such access would violate the encapsulation of g_A (i.e., would constitute access to the design of g_A rather than to its interface); if g_H were contained within g_A, its type specification couldn't link anything to any attribute of g_A because objects of g_H's type need not necessarily be contained only within objects of g_A's type.

For simplicity, only a single link to a given method, argument, or result is permitted. If multiple links were allowed it would be necessary to impose some sort of arbitration scheme on an already complex plan in order to resolve conflicts arising from concurrent attempts to change the value of a given method or argument instance variable in different ways. Most real-world implementations of software built on the object model automatically avoid the problem by virtue of the Von Neumann architecture of the hardware they run on: since nothing truly happens concurrently anyway, no conflict is possible. In effect, the constraints on linkage achieve this arbitration structurally without restricting parallel execution: a given object can only be activated (its method instance variable set) from one other object, either the one that immediately contains it or one of its peers.

By itself, however, this arbitration is too severe, Objects that exist merely as repositories for the data of their immediate containers can reasonably be insulated from activation by any peer, but in many cases an object may need to serve data to (or accept data from) any number of its peer objects, Some more elaborate arbitration scheme is still needed; we propose one that is constructed from objects rather than built into the model itself,

The scheme essentially queues peer-to-peer communication by adding more peer objects, of two types: port and router. The type "port" is defined by **(port, (E, I, N, D, L, A, Z))** where:

E = {receive, erase)

I = {(receive, (τ, τ)), (erase, (τ, τ))}

N = ϕ

D = ϕ

L = ϕ

A = ϕ

Z = {(({(m, receive), (*ad*, x), (*rd*, y), (*s*, 7,) , a),

{(m, erase), (*ad*, e), (*rd*, x), (*s*, z)]) a $\in \Sigma$, x $\in \Lambda$, y $\in \Lambda$, z $\in \Lambda$}

$\cup$ {(({(*m*, erase), (*ad*, x), (*r*, y), (*s*, z)}, a),

{(*m*, *e*), (*ad*, x), (*r*, *e*), (*s*, z)}) : a $\in \Sigma$, x $\in \Lambda$, y $\in \Lambda$, z $\in$ A }

When a port's "receive" method is initiated, it simply copies its argument to its result and changes method to “erase”; its “erase” method merely erases the port’s result attribute and deactivates the port, enabling it to receive another string.

The formal definition of a message router for a given object system would be too lengthy to present here. Informally, for each peer object in the system to be connected through the routing mechanism (called a “client”), the router has three τ attributes: one is linked to the *rd* attribute of an “inbound” port (a peer of the router) that the client uses to send messages to the router, while the other two arc linked to the method and *ad* argument of an “outbound” port that the client uses to receive messages from the router; one inbound port and one outbound port are provided for each client, The router also has a “message queue”, a τ attribute whose value at any moment is the concatenation of all messages not yet delivered to their destinations. The client, in turn, has attributes linked to the method and *ad* of its inbound port and another attribute linked to the *rd* attribute of its outbound port.

When the client wants to activate a peer client it puts in the *ad* attribute of its own inbound port a message, a string formed by concatenating the name of the client the message is to go to, the name of the client sending the message, some string that uniquely identifies the message itself (for reference in handling a reply to this message), the name of the method to activate, and a string representation of the argument to pass to it; the client also changes the method of that port to “receive”. (The client must wait briefly if the port is currently active.) The port copies the message to its result and erases its method so that it can receive another message. Meanwhile, on every transition the router object:

1. Appends to its message queue all non-null messages in the *rd* attributes of all of its inbound ports.

2. For each inactive outbound port, puts into the port’s *ad* attribute the first message in the message queue destined for the corresponding client, removes that message from the queue, and changes the method of the port to "receive".

All clients are perpetually active. On every transition, in addition to whatever else it is doing, every client appends to its own queue of unhandled messages the contents of the *rd* attribute of its outbound port. When a client finishes processing a given message it begins processing the next one in its queue or goes into a state in which it is waiting for a message to arrive.

Note that this sort of object interaction readil y supports parallel processing in arbitrary object distribution patterns. It is in general “slower” but more powerful than raw instance variable linkage: transmission of any message consumes at least 4 “clock ticks” (state transitions) rather than 1, but the device enables any object to communicate not just with one single other object but with all of its peers, all the objects it immediately contains (a pair of ports could additionally be provided to enable the container of the message router to communicate with all of the router’s clients), and the object that immediately contains it. Note also that the arguments and results of ports could bc of types other than τ; for example, a port could have one argument for each type of argument T that could be passed from onc client to another, where the design of this port argument includes τ attributes for the message ID, the names of destination and source objects, and the method name, together with a T attribute for the argument of the message. This would enable static type checking with reference to the receiving clients’ interfaces. (The simpler approach was presented above for the sake of brevity.)

We suggest that the differences between peer-to-peer message exchange as described here and direct communication with immediately contained objects are so profound that they should be referred to by different terms. We propose to call the former “message passing” and the latter “method invocation”. This diverges somewhat from current practice in most object technology implementations, which normally use the terms interchangeably to refer to what are, in fact, function calls. We believe, though, that the use of the phrase “sending a message” in the object model should agree as closely as possible with the everyday notion of composing a message -- a data structure with an integral, independent existence -- and transmitting it through some sort of communications medium.

inheritance

Apart from containment of objects of various types (which models “has a“ relationships), the major mechanism for code re-use in object-oriented software is inheritance, the modeling of “is a“ relationships by defining a ncw type as an extension of one that was previously defined. In the mode] presented here, each member of a characteristic ancestry is a single line of inheritance; multiple inheritance results from defining multiple ancestors.

The aggregate personality of each object is fixed at the time the object system begins to operate, However, multiplicity of personae of fluctuating influence within a single ancestor would enable various types of dynamic multiple inheritance. As promised earlier, we defer a full description of this mechanism for a later paper, but briefl y:

> We can relax the constraint on ancestors to allow multiple personae.
>
> We can stipulate that every type’s implicit design additionally contains one “membership” attribute for each persona in the t ype’s aggregate personalit y, and we can make transitions inherited from the personae conditional on the values of those attributes.
>
> We can add transitions that alter the values of membership attributes in response to changes in other attributes’ values.

In short we claim that dynamic inheritance schemes (as in [6]) can be clearly articulated within the framework of this model just by including all potential personae in the ancestry of a t ype and letting their influences vary over time. We suspect that the notion of predicate classes [7] can be similarly formalized, by representing predicates as sets of resolved state space elements and adding transitions that alter the values of membership attributes in response to those predicates. Moreover, wc plan to consider the impact of allowing membership attributes to have “fuzzy” values, potentially making a single object partially a member of multiple mutually exclusive classes at once; this may be a straightforward way to conceive of “fuzzy” object-oriented software.

The inheritance mechanism modeled here does differ in several significant ways from that which is implemented in many object-oriented programming languages,

1. Citation of the same type in multiple ancestors -- that is, repeated inheritance -- always results in the “sharing” of the redundantly cited types and the merging of their attributes. We believe this best reflects the manner in which such conflicts are resolved in real-world classification schemes. For example, if we say that a money-market account is both an investment and a checking account, both of which are regarded ›y a bank as credit accounts, we would expect a money-market account to have only a single account number and a single owner, even though every credit account has a distinct number and owner.

   Note that this mechanism makes it difficult to use multiple inheritance to establish multiple involvements in the same type of relationship, e.g. to model a Taxpayer’s N employment situations by inheriting N times from an Employee type rather than containing N references to Position objects. While this could bc considered a defect, we regard it as a feature; we believe it encourages more natural and familiar modeling structures.

2. The sets of qualified attribute names contributed by all members of a type’s aggregate personality y must be disjoint; attribute name clashes are simply proscribed, Again, this seems consistent with the real world. If checking accounts are charged services fees at some rate per check written and investments are credited with interest

at some rate per dollar of minimum balance, we would expect different terms to be used for these two different “rates” -- perhaps perCheckCharge and interestRate.

3. Since the method lists of all members of a t ype’s aggregate ancestry must likewise be disjoint, there is no notion of the overriding of methods. That is, a type maybe regarded as a refinement or extension or specialization -- a “subtype” -- of its ancestor type(s), but no conflict between the methods of a subtype and those of its ancestors is permitted. Once again, we believe this best reflects our real-world expectations of classification schemes, For example, a snake is a kind of reptile; a python is a kind of snake. We could say that a snake is a legless carnivorous reptile that kills its prey by injecting it with poison through a bite, and that a python is a snake except that it kills by constriction rather than poison, but we don’t. Instead, wc say nothing about predation when we talk about snakes in the abstract (though we do still say they’re legless carnivorous reptiles), we subclassify snakes into poisonous and nonpoisonous varieties, and we say pythons are nonpoisonous snakes. This latter approach ensures that we know something reliable just from knowing how pythons are classified, and that this classification doesn’t tell us something unreliable that we can only get straightened out by knowing specific information about pythons. Our definition of subclass ensures that if X is a subclass of Y then everything that is true of the behavior of Y objects is also true of X objects <u>without</u> <u>exception</u>.

4. Many implementations of object technology regard an object as an “instance” of a class. In the context of our model one might say that an object is an instance of a single, specific type, which completely defines its structure and behavior (that is, adopt the terminology “instance of type T“ as an alternative to “object of type T“). One might also observe that a given object is a **member** (not an instance) of the class correspond-

ing to the object's type, but it may also be a member of many other classes. By the same token, it may be impossible for any member of a given class to be an instance of the type of the same name, because that type may not be viable. For example, a given object might be a" 1985 Toyota Tercel hatchback" but it is also an "automobile" and an "industrial product"; each of these classifications is correct, but only the first -- the type -- provides enough information to instantiate the object. "Industrial product" is a non-viable type, in that no one knows how to make one in the absence of more detailed specifications.

Instance variables defined in the characteristic designs of an object's inherited personae are encapsulated just as are those of the object's own characteristic design, but the invocation of a method inherited from an ancestor is different from the invocation of the methods of contained objects. Because the inherited personae of an object g_A, of type T, are not contained within g_A but arc instead integrated with g_A (that is, they are other aspects of the nature of g_A itself, not distinct objects), they do not have separate method variables. Object g_A performs method Xyz inherited from its ancestor type Q mere] y by setting the variables of arguments as necessary and then changing g_A's own method to Xyz; although the characteristic transition function of T (g_A's type) docsn't know how to respond to Xyz, the characteristic transition function of g_A's Q ancestor does.

Typically, Xyz will bc invoked from within some other method of g_A which should be resumed when Xyz terminates, This continuity is made possible by the method stack instance variable. Before switching its method to Xyz, g_A can prepend the current method name, with some sort of delimiter, to the method stack; when Xyz terminates it can set the values of result instance variables, then pop the prefix of the method stack and change method to this prefix rather than e, putting the object back in the state it was in before Xyz was called (aside from changes resulting from Xyz). When the logic of a method is modularized by some mechanism such as a "phase" attribute, the current phase is automatically resumed when the method resumes; the calling of Xyz cannot have affected the phase because phase is private to the characteristic design of T.

The method stack also makes it possible for a descendant to invoke ancestor capabilities that arc not offered to the object's peers or immediate container (in C++ terminology, "protected" methods). Although all methods are public, a given method of an ancestor type could inspect the prefix of the method stack to determine how to proceed: if that prefix was not, for example, a phase number prepended to the stack by a descendant after the descendant's own method name was prepend-cd, the ancestor method would realize that it was being invoked by some other, unauthorized object and the method could simply terminate. Since the method stack is private to the members of the aggregate ancestry of an object, no external object can affect its contents.

Just as a descendant can invoke a method of onc of its ancestors, an ancestor can invoke a method that must be provided by its descendant -- a "pure virtual method". Types whose aggregate transition functions include such pure virtual method invocations are non-viable types (in C++ terms, "abstract classes"). Such types can bc the ancestors of subtypes, but by definition no attribute of any design can bc of a non-viable type, and a non-viable type cannot be used to construct an object system.

Pol ymorphism

Object systems built on this mode] can directly exhibit some varieties of polymorphism [8] but not others:

> Overloading: Clearly, two different types may declare methods of the same name (provided the two types are not in the same aggregate ancestry). Initiating this method within objects of the two types may yield very different results.
>
> Coercion: An object is always of a single, invariant type; the model includes no

notion of “coercing” an object oft ypc X into type Y to satisfy the interface of a method, and clearly an argument value of type X cannot be presented directly to a method M that obtains its input from an *a* instance variable of type Y. If X is a subtype of Y, however, it should be straightforward to extract the values of the Y instance variables embedded in the X argument value and load them into the Y argument. The same would be true of argument values of types W, V, etc., provided all of these types were subtypes of Y. A language implementation that, in effect, performed this extraction and loading transparently would have provided a coercion facility.

inclusion polymorphism: Two objects may be instances of different types but members of the same class, defined by a common ancestor type that has in its interface a given method. Initiating that method within each of the two objects may yield either the same result or different results, depending on the states of the objects and on wheth-er or not the method is virtual, i.e., declared in the ancestor but defined (perhaps in different ways) in the descendants. Alternatively, given multiplicity of persona within an ancestor, the execution of a given method by a given object might change over time as the influences of its personae varied.

Parametric polymorphism: In this model, the names of all methods are unique and are therefore sufficient for the purpose of method selection. However, a language implementation of the model might well permit different methods taking different types of argument to have the same name externally and make them internally unique by appending argument type labels to the name.

instantiation, destruction, persistence, garbage collection

Note that nothing in the exposition above indicated how objects might be created and deleted. Thc reason for this is that the creation and deletion of objects would violate the model: by definition, they already exist -- one pcr member of the aggregate design of the type specifying the object system they belong to.

Although this concept is somewhat foreign to contemporary object technology, it’s not completely unfamiliar. In the real world we may say that such things as loyalty, Huckleberry Finn, and the assassination of the Archduke Ferdinand “exist”: even though they occupy no physical space they have the capacity to shape events. Similarly, a temporary linked list in an object system may be an object that no longer occupies physical memory, but it still “exists” in the source code for the program. The determinism in software is what makes it testable, and in any truly deterministic system whatever is possible is inevitable. Everything that could exist does (in some sense) already exist, although at any particular moment it may have no physical manifestation,

The determinism of the model presented here maps easily to practical software determinism: if wc merely add the attribute “physical memory location” to the design of an object and leave that attribute null until wc “create” the object (in a practical implementation, allocate physical memory to it) -- and reset it to null when we “destroy” the object -- we retain theoretical consistency without at al 1 sacrificing compatibility with implementation strategies,

This leaves open the questions of how to allocate physics” memory and how and when to release it. Since theoretically all objects in an object system always exist, “object persistence” and “garbage collection” are strictly implementation issues. Ideally all objects would always be persistent except when they were explicitly “destroyed” for some <u>functional</u> reason, at which time the release of the space they occupy would be straightforward. Restrictions on persistence ahd complex mechanisms for the implicit, automatic destruction of objects when no more references to them remain are an unfortunate artifact of hardware limits.

Metaclasses, Reflection

Throughout its operation, the composition of an object system produced according to this model remains fixed. This unquestionably limits the flexibility of such a system but, we believe, not fatally.

The population of an object system (the number of objects of all types) is fixed by its aggregate name space and design. However, as discussed above, the practical implementation of any such system can dynamically allocate and free the physical memory occupied by objects in complete accord with the model.

The heritage of every object is fixed by its aggregate ancestry. However, varying the influences of various personae over time -- including those whose original influence may have been zero -- can have effects that would be indistinguishable from the dynamic acquisition of ancestors. (A specific implementat ion of variable persona influence could, of course, defer allocation of physical space for a persona's instance variables until the persona's influence becomes non-zero.)

Finally, the structure and behavior of each individual object are fixed by its characteristic method list, interface, design, linkage, and transition function. Since some object technologies provide "metaclass" objects to enable class or type definitions to change while software is executing (as discussed in [9]), it would seem that a DFA cannot model all object system implementations. We suggest, though, that type redefinition in fact occurs in a "development" universe that is external to the object universe of the operating object system -- i.e., that it is within the scope of the "object-oriented software engineering" paradigm but out of the scope of the object model itself.

For this reason, the model presented here includes no notion of metaclasses or "class methods", “class data”, etc. By definition, an object is not a class, a class is not an object, and a type is neither; there is no infinite regress of objects and classes, because classes and types are not instantiation of anything else.

This is not to say that runtimc modification of class definitions is undesirable or impractical, only that it is an implementation feature that is external to the object model. And though it is external to the model, it is not incompatible with it. The model is in no way violated if we “stop the tape” of input symbols of an operating object system X, putting the DFA in a final state (i.e., whatever its state currently happens to be), We can then copy all relevant state elements from that final state into the starting state S_w of a new object system W that is a modification of the one that just stopped (“relevant” elements being those ordered pairs (n, v) such that $(n, \tau) \bullet D_w$); set to e the values of all (n, v) in S_w such that $(n, \tau) \bullet D_w$ but $(n, \tau) \not\subset D_x$; and then start W. Since this happens while no input symbols arc being processed, in terms of the model it doesn’ even consume any time.

Conformance to object software principles

We believe the model presented here generally conforms to widely accepted principles of good object system form. For example:

It satisfies B. Meyer’s “open/closed” principle [10], in that types are open for extension but closed for purposes of inheritance and containment. In fact, because a subtype can extend the functionality of an ancestor type but cannot modify [override] it, the “closed-ness” of types is more severe than that which is advocated by Meyer. The intent is the converse of Meyer’s reusability requirement [11]: not only should a feature be moved as far up in the inheritance hierarchy as possible, to enable the broadest possible sharing by descendant classes, but also it must be moved as far down in the hierarchy as necessary in order to ensure its applicability to all descendant classes without exception.

It conforms to the Liskov substitution principle [12]: if U is a subtype of T, then an object of type U may bc substituted for an object of type T in any object system X without changing the behavior of the system, This is due to the constraint that the method lists of all members of an aggregate ancestry must be disjoint: since U's transition function includes all T transitions implementing the methods in T's interface, and since the behavior of T objects in X must have been limited to the methods in that interface, whenever a replacement object of typc U in X is activated it must be by some request to perform some method of T, which will be handled by the inherited T transitions.

If message passing is used for peer-to-peer object interaction, then it enforces compliance with the letter, if not necessarily the spirit, of the Law of Dcmcter [1 3]: the meth-ods defined for a client object g_A of a given type can directly invoke the methods only of hc objects contained within g_A, the argument to the method (since all arguments are actually objects contained within g_A), and g_A itself. Functionally, however, g_A can still send messages to any number of peer objects of various types, so the decision on how zealously to comply with the Law remains mostly the responsibility of the programmer.

Application

One usc to which a formal theory of objects might initially be put is evaluation of the many object-oriented programming languages currently available to the developer. In the context of the model discussed here, an object-oriented programming language is simply a notation for specifying t ypcs. A language is "complete" if it can bc used to specify any type whatsoever (given that the type's A is compatible with the language); it is "convenient" to the extent that it makes type specifications undertaken for a given purpose easy to write and easy to read. The compiler for such a language is "sound" if the operations of the executables it produces accurate] y predict the state transitions of the corresponding types, and vice versa; it is "efficient" to the extent that it compresses the DFAs **produced by the language into** forms that can be executed on physical computers within cost and time constraints. (As the example definition of "number" above suggests, standard set notation can be regarded as a complete but relatively inconvenient object-oriented programming language for which no compiler exists,)

Conclusion

The motivation for this work was the need for some formal theoretical framework that would facilitate the clear articulation of object model concepts. Much in the model statement presented here may be open to challenge. Nonetheless, we hope it will contribute to the development of that framework.

Acknowledgement

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

Affiliation of Author:
Scott Burleigh
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, M/S 301-270
Pasadena, CA 91109
burleigh@puente .jpl.nasa.gov

[1] Understanding Object-Model Concepts / Position Papers for 00PSLA '93 Workshop #19, D. W, Embley, cd., Provo (UT), Brigham Young University Computer Science Department, 1993, p. 1.

[2] A. Snyder, "The Essence of Objects: Concepts and Terms", IEEE Software, January 1993, pp. 31-42.

[3] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, Designing Object-Oriented Software, P T R Prentice Hall, Englewood Cliffs (NJ), 1990, pp. 17-36.

[4] Y. Wand, "A Proposal for a Formal Model of Objects" in Object-Oriented Concepts, Databases, and Applications, W. Kim and F. Lochovsky, eds. , ACM Press, New York, 1989, pp. 537-559.

[5] H. R. Lewis and C. H. Papadimitriou, Elements of the Theory of Computation, Prentice-Hall, Englewood Cliffs (NJ), 1981, p. 51.

[6] D. Ungar and R. B. Smith, "SELF: The Power of Simplicity" in OOPSLA '87 Conference Proceedings, ACM Press, New York, 1987, pp. 227-241.

[7] C. Chambers, "Predicate Classes" in Proceedings of ECOOP '93, Springer-Verlag, Kaiserslautern (Germany), 1993, pp. 268-296.

[81 L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", ACM Computing Surveys, ACM Press, New York, 1985, pp. 471-522.

[9] G. Kiczales, J. des Rivieres, and D. G. Bobrow, The Art of the Metaobject Protocol, MIT Press, Cambridge, 1992.

[10] B. Meyer, Object-Oriented Software Construction, Prentice-Hall, Hertfordshire (U.K.), 1988, p. 229.

[11] Ibid. p. 230.

[12] B. Liskov, "Data Abstraction and Hierarchy", SIGPLAN Notices, vol 23 no. 5 (May 1988), p. 25.

[13] K. Lieberherr and I. Holland, "Assuring Good Style for Object-Oriented Programs", IEEE Software, September 1989, pp. 38-48.